PharmaSUG 2019 - Paper PO-22

In the Style Of David Letterman's "Top Ten" Lists, Our "Top Ten" PROC SQL Statements To Use in Your SAS Program

Margie Merlino, Janssen Research and Development

ABSTRACT

One of the challenges in using PROC SQL is the notion that a programmer must first immerse him or herself in the education of Structured Query Language (SQL) before they can use PROC SQL. Certainly, there is syntax that a programmer must adhere to for successful execution a PROC SQL statement, but we propose that there are many simple and some not so simple statements that can be used without a robust knowledge of all the intricacies of SQL. What follows is a "Top Ten List" ala David Letterman's popular feature. We list ten scenarios and a PROC SQL example that addresses that scenario and can be used to manipulate the SAS data and display a result or provide an output dataset for further manipulation. Our goal is to provide code examples that a programmer can easily copy/paste into their own SAS program.

INTRODUCTION

When I want help with a technique in a programming language I look for code examples. My mantra is, “Don’t tell me about it, show me.” When I was new to java programming I landed on a fantastic website, “JavaRanch - A friendly Place for Java Greenhorns”. The name alone was whimsical enough to draw me in, but what I liked most was the wealth of examples to copy/paste into my java programs.

I have not found a comparable “greenhorn” site for PROC SQL so I’ll contribute what I’ve found to be handy uses for PROC SQL that use straightforward constructs. I think of PROC SQL as one tool in my SAS programming tool chest. It’s there, so why not use it? It’s my hope that you can find sample code in this paper to include in your own SAS tool chest.

What follows is my list of favorite or “Top Ten” uses for PROC SQL. Perhaps the explanations and examples will pique your interest enough to give them a try!

INITIALIZE A SINGLE MACRO VARIABLE

It is very straightforward to create a macro variable in PROC SQL. Let's start with two examples in which we're initializing a single macro variable with a single value. In the first example we're hard-coding the value to be stored in the macro variable, much like a "%let" in SAS. In the second example we're applying a function to a lab (LB) variable to create the value stored in the macro variable. Note that in the first example, we're not using values from the my_data dataset but the syntax requires the "from <dataset_name>" part of the construct:

```
proc sql;
    select 'Lupus' into :disease from my_data;
quit;

proc sql;
    select mean(lbstresn) into :mean_alt from sdtm.lb where lbtestcd='ALT';
quit;
```

Next, we'll initialize a single macro variable with multiple values concatenated with a character, with the result displayed below:

```
proc sql;
    select distinct(therapeutic_area) into :all_ta separated by "|" from
    all_studies;
quit;

%put &=all_ta;
ALL_TA=CV/M|IDV|Immunology|MAF/EP|Neuroscience|Oncology
```

INITIALIZE MULTIPLE MACRO VARIABLES WITH A SINGLE PROC SQL STATEMENT

To initialize multiple macro variables in one PROC SQL statement with hard-coded values:

```
proc sql noprint;
    select 5,32,10,22,18,36,123
    INTO :cvm_denom, :idv_denom, :imm_denom, :neuro_denom, :onc_denom,
    :maf_denom, :tot_denom from dummy_data;
quit;
```

You should make sure that the number of values in the select match the number of macro variables that you're populating. To many or too few and you'll get a warning in your log file.

To further illustrate how to initialize multiple macro variables at one time, let's consider one of our previous examples:

```
proc sql;
    select distinct(therapeutic_area) into :all_ta separated by "|" from
    all_studies;
quit;
```

Instead of jamming all the values from our 'select distinct' into one variable, we'll break those into multiple macro variables. The results are displayed below the PROC SQL:

```
proc sql;
    select distinct(therapeutic_area) into :ta1 - :ta7 from all_studies;
quit;

%put _USER_;
GLOBAL TA1 CV/M
GLOBAL TA2 IDV
GLOBAL TA3 Immunology
GLOBAL TA4 MAF/EP
GLOBAL TA5 Neuroscience
GLOBAL TA6 Oncology
```

No warnings are displayed in the log if you provide too many or two few macro variables in your "into" statement.

APPLES AND ORANGES – MERGING ON UNLIKE VARIABLE NAMES

In most of my other examples I use SDTM variable names which are nice and standardized, but we don't always have the luxury of that "standardized" condition in our data. A handy feature in PROC SQL when working with less standard data is the ability to merge on two different variable names. Let's suppose your subjects are identified differently across the datasets you're merging:

```
proc sql;
    create table history as
    select a.*, b.*
    from med_hist as a, sdtm.dm as b
    where strip(a.subject) = strip(b.usubjid);
quit;
```

STACK 'EM UP – THE UNION STATEMENT

The union statement in PROC SQL is pretty much the same as a SET statement in a data step if all you're doing is stacking datasets as-is. But if you're doing multiple variable manipulations such as creating and/or renaming variables plus doing inline functions to the datasets while stacking then consider the union statement since it may be less wordy than using a datastep. In the example below I'm combining adverse event (AE) datasets from multiple studies:

```
proc sql;
create table all_ae as
    select 'Diabetes' as disease, usubjid, catx('/',strip(aeterm),
    strip(aedecod))as ae_combine, aehlgt as group_name from diab.ae
    union
    select 'Arthritis' as disease, usubjid, catx('/',strip(aeterm),
    strip(aedecod))as ae_combine, aehlt as group_name  from arth.ae
    union
    select 'Migraine' as disease, usubjid, catx('/',strip(aeterm),
    strip(aeterm))as ae_combine, aehlgt as group_name from migr.ae
    ;
quit;
```

DOING DOUBLE-DUTY - CREATE AND SORT DATASET IN ONE STEP

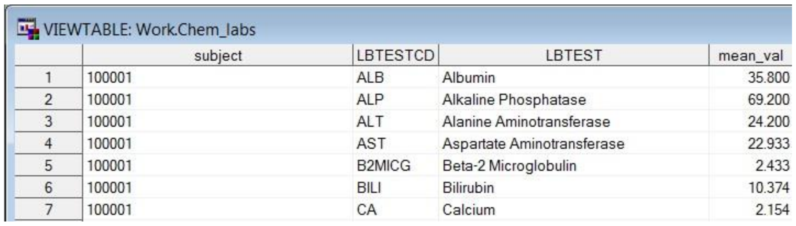
Instead of having both a data step and a PROC SORT, the PROC SQL allows you to create and sort a dataset using the “order by”:

```
proc sql;
    create table chem_labs as
    select lb.*, dm.sex, dm.age
    from sdtm.lb as lb, sdtm.dm as dm
    where lb.usubjid = dm.usubjid
    and lbcat='CHEMISTRY'
    order by lb.usubjid, lbtestcd;
quit;
```

TOTALS AND SUB-TOTALS

The “group by” in a PROC SQL statement allows you to total numeric results at various levels and this does double-duty as a sort too. Below we’re subtotaling mean chemistry lab results by subject and lab test:

```
proc sql;
    create table chem_labs as
          select substr(usubjid, 17,6) as subject, lbtestcd, lbtest,
          mean(lbstresn) as mean_val format=10.3
          from sdtm.lb
          where lbcat='CHEMISTRY' and lbtestcd <> 'LBALL' and not
          missing(lbstresn)
          group by subject, lbtestcd, lbtest;
quit;
```

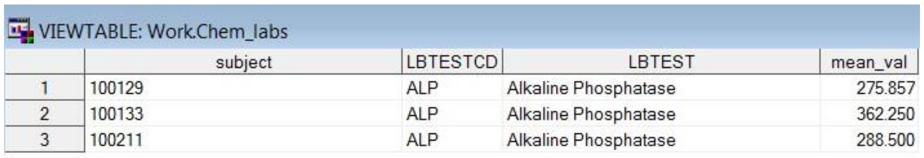

VIEWTABLE: Work.Chem_labs

	subject	LBTESTCD	LBTEST	mean_val
1	100001	ALB	Albumin	35.800
2	100001	ALP	Alkaline Phosphatase	69.200
3	100001	ALT	Alanine Aminotransferase	24.200
4	100001	AST	Aspartate Aminotransferase	22.933
5	100001	B2MICG	Beta-2 Microglobulin	2.433
6	100001	BILI	Bilirubin	10.374
7	100001	CA	Calcium	2.154

MORE DOUBLE-DUTY - FILTER ON FUNCTION RESULTS

Using the PROC SQL query above, we can filter on the results of a function using the “having” clause:

```
proc sql;
    create table chem_labs as
    select substr(usubjid, 17,6) as subject, lbtestcd, lbtest,
mean(lbstresn) as mean_val format=10.3
    from sdtm.lb
    where lbcat='CHEMISTRY' and lbtestcd = 'ALP' and not missing(lbstresn)
    group by subject, lbtestcd, lbtest
    having mean_val > 275;
quit;
```

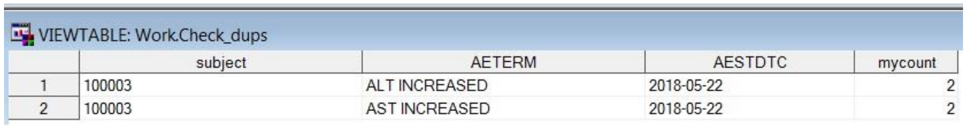

VIEWTABLE: Work.Chem_labs

	subject	LBTESTCD	LBTEST	mean_val
1	100129	ALP	Alkaline Phosphatase	275.857
2	100133	ALP	Alkaline Phosphatase	362.250
3	100211	ALP	Alkaline Phosphatase	288.500

FIND THOSE DUPLICATES

Ok, this is another flavor of filtering described above, but it’s a good use of the “having” statement. I’ve used this technique when working on dataset quality control (QC) activities. Let’s look at an example in which I’m checking for duplicate adverse events (AE) reported on the same date. These may be legitimate entries since it’s entirely possible to have the same adverse event occur multiple times on one day for a subject, but the results from this query will narrow down my investigation:

```
proc sql;
    create table check_dups as
    select substr(usubjid, 17,6) as subject, aeterm, aestdtc, count(*) as
    mycount
    from sdtm.ae
    group by subject, aeterm, aestdtc
    having mycount > 1
    ;
quit;
```


VIEWTABLE: Work.Check_dups

	subject	AETERM	AESTDTC	mycount
1	100003	ALT INCREASED	2018-05-22	2
2	100003	AST INCREASED	2018-05-22	2

CARTESIAN JOIN

In any beginner's SQL class we're told that Cartesian joins are bad. We're told if your SQL code results in a Cartesian join it's the result of a mistake in your code. However, there may be situations in which you actually do want to merge all the rows in a table with all of the rows in another and it's difficult, at best, to do that in SAS. PROC SQL will allow it, albeit with a message in the log that you've performed a Cartesian join.

Here we combine the contents of two datasets together without any qualifiers. In this example the second of the two datasets in the Cartesian join (mydate) contains just a single row of data so the resulting dataset will have the same number of rows as the first dataset (sdtm.ae):

```
proc sql;
    create table mydate as
          select distinct(put(today(), is8601da.)) as updte format=$10.,
          "Refresh RAVE data" as upd_reas format=$50. from dummy_ds;

          create table myae as
          select a.*, b.*
          from sdtm.ae as a, mydate as b;
    ;
quit;
```

Realistically you would do the above activity all within the second create statement, but it provides the construct for the most extreme example of a Cartesian join without taxing our SAS resources. Let's look at a query that's not a Cartesian join, but we're taking advantage of PROC SQL by not fully qualifying.

Here we're qualifying the merged dataset solely on the usubjid and a date range. We've prepped the input datasets, AE and LB, to consider only those related to liver-related events and a specific lab test because for each AE we want to merge every "ALP" lab test with a date during the adverse event. Consequently, for each of our AE's we'll join on any lab test that meets the date criteria for that subject:

```
proc sql;
    create table ae_liver as
          select * from sdtm.ae
          where upcase(aehlt) = 'LIVER FUNCTION ANALYSES';

    create table lb_alp as
          select * from sdtm.lb
          where lbtestcd = 'ALP';

    create table liver_aelb as
          select a.*, b.*
          from ae_liver as a, lb_alp as b
          where a.usubjid = b.usubjid
          and ((aestdtc <=lbdtc <= aeendtc));
quit;
```

A SUB(QUERY) IS NOT A HOAGIE

That's a little shout-out to my hometown and the 2019 PharmaSUG host city of Philadelphia where we take our hoagies seriously.

This is more of an intermediate topic, the PROC SQL subquery, and it can be used to further qualify the results of a PROC SQL query. I found a good description of the various kinds of subqueries in a paper from a PharmaSUG conference several years ago, "https://www.pharmasug.org/proceedings/2013/PO/PharmaSUG-2013-PO15.pdf".

The example below is a non-correlated subquery which means the subquery select, which is within parentheses, can be executed independently of the main select:

```
proc sql;
    create table high_wbc as
    select a.* from cll.ae as a
    where upcase(aehlgt) = 'WHITE BLOOD CELL DISORDERS'
    and a.usubjid in (
    select distinct b.usubjid from cll.lb as b
          where lbtestcd = 'WBC'
          and lbstresn > 20
          and index(b.visit, 'UNSCH') > 0);
quit;
```

This will create one row in the high_wbc dataset for each row in the first or "outer" query that meets criteria of the subquery select. Note that the "distinct" in the subquery could be removed and create the same resulting table from the overall query. However, the "distinct" in the subquery creates a smaller result set for the outer select to interrogate.

CONCLUSION

PROC SQL has some very handy uses, not "instead of" data steps but "in addition to" data steps.

REFERENCES

Vemuri, Pavan. 2013. "SQL SUBQUERIES: Usage in Clinical Programming." Proceedings of the 2013 PharmaSUG Conference. Available at "https://www.pharmasug.org/proceedings/2013/PO/PharmaSUG-2013-PO15.pdf".

https://javaranch.com/.

https://tv.avclub.com/.

http://herbalcare.info.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Margie Merlino
Janssen Research and Development
mmerlin2@its.jnj.com